
LearnDelphi.org
Coding Boot Camp
Session 34 [redacted]
Delphi and functional programming
using anonymous methods
[redacted]

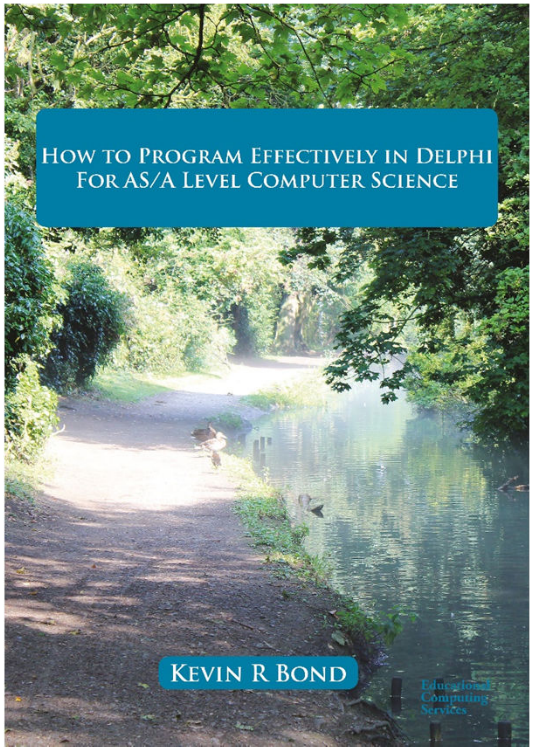
Coding Boot Camp 2022

How to Program Effectively in Delphi for AS/A Level Computer Science

By Kevin R Bond

Available in both print and pdf formats from
https://www.educational-computing.net

Main site for information
http://www.educational-computing.co.uk



Length =1200 pages.
Suitable for all levels from beginner to advanced developer

Overview

1. Background

2. Concepts - Function, Function type, Function application,
   Partial function application, Function composition,
   Lambda calculus

3. Higher-order functions – Map, Reduce, Filter

4. Closure.

Programming paradigms

- LOOSELY SPEAKING THIS REFERS TO THE EXECUTION MODEL AND HOW THE CODE IS ORGANISED.
- THERE ARE TWO MAIN PARADIGMS:
  - ❑ IMPERATIVE
  - ❑ DECLARATIVE

Execution model and organisation closely follows the way the underlying machine operates – one statement at time, statements executed in sequence, statements/operations change the state of variables/objects, execution has some effect on a memory store.

In declarative programming the programmer supplies the what (what information is required) and the how is left to the language to figure out. Execution and organisation doesn't follow the way the underlying machine operates. This reduces the brain-to-code distance by supporting programming at a very high level.

Programming paradigms

- Imperative – the world is sequential, time marches forward because of the 2nd Law of Thermodynamics $\Delta S \geq 0$

  - ❑Procedural – Pascal, mutable state

  - ❑Object-oriented – Object Pascal, encapsulated mutable state

  - ❑Event-driven – Delphi

Programming paradigms

- Declarative – time independent
  - Logic – e.g. Prolog, programs consist of facts and rules, uses deductive reasoning to solve programming problems. Relies heavily on pattern matching and recursive backtracking
  - Functional - programs are constructed by applying and composing functions. Emphasises the evaluation of expressions rather than execution of commands. No loops instead uses recursion (tail recursion)
    - Pure – uses functions (or expressions) which have no side effects (memory or I/O), and which always return the same output, given the same input.
    - Impure - side effects, e.g. I/O is used

Logic + Functional = SQL

Logic + Functional = SQL

PascalABC.NET

```
begin
  var s := ' hello  aha paap   zz ';
  s.ToWords.Where(w -> w.Inverse = w).OrderBy(s->s.Length).Println(',');
end.
```

SQL

```
Select CustomerSurname
  From Customer
    Where Customer.CustNo In (Select CustNo
                                From Orders
                                  Where Orders.PaymentMethod = 'Visa')
    OrderBy CustomerSurname
```

Functional programming in Delphi using anonymous methods

```
DbleArr2.Map(Function(ValueInArray : Double) : Double
               Begin
                 Result := ValueInArray + 5;
               End).Filter(Function(ValueInArray : Double; Index : Integer) : Boolean
                             Begin
                               Result := (Index Mod 2) = 0;
                             End).ForEach(Procedure(Var ValueInArray : Double)
                                            Begin
                                              Write(ValueInArray : 2 : 0)
                                            End);
```

Programming paradigms

- Multi-paradigm – mixes procedural, OOP and functional to various degrees
  e.g. Object Pascal, C++, Java, JavaScript, C#, Scala, Visual Basic, Common Lisp, PHP, Python

What are anonymous functions

- Anonymous functions originate in the work of Alonzo Church in his invention of the lambda calculus , in 1936, in which all functions are anonymous.

$$\lambda x \to x * x$$

- An anonymous function is a function definition that is not bound to an identifier.
  In Haskell, this would be written as

```
\x -> x*x          (\x -> x*x) 6
```

https://replit.com/languages/haskell

- Lambda expressions are really just anonymous functions in a concise form
  (A Lambda expression is a name, e.g. x, or a function, e.g. \x -> x*x, or a function application, (\x -> x*x)).
  (A Lambda calculus is a system for manipulating Lambda expressions)

```
Program PolynomialSumProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils, System.Math;
Type
  TPolynomialFunction = Reference To Function(x : Double) : Double;

Function Sum (n : Integer; f : TPolynomialFunction) : Double;
  Begin
    Result := 0.0;
    For Var i := 1 To n Do Result := Result + f(i);
  End;
Begin
  Writeln('Sum = ', Sum(3, Function(x : Double) : Double
                            Begin
                              Result := x;
                            End) : 16 : 12);
  Readln;
End.
```

1 + 2 + 3

NB: TPolynomialFunction is not a function pointer, viz.

```
Type
  TFunctionVar = Function (y : Integer) : Integer;
Var
  FunctionVar  : TFunctionVar;
```

NB: Neither is it a function reference for a function contained in an object, an instance of a class, viz.

```
Type
  TFunctionVar = Function(x : Double) : Double of Object;
```

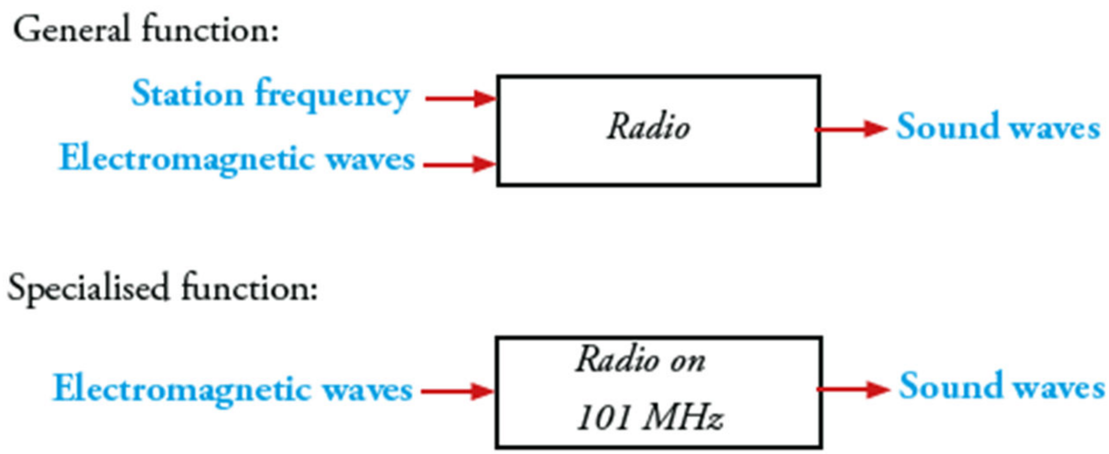
TPolynomialFunction is actually an Interface type definition

Compiler takes the TPolynomialFunction interface definition, creates a class that implements this interface and then creates an object containing the function f

Try this in Delphi 10.4.2

Partial function application

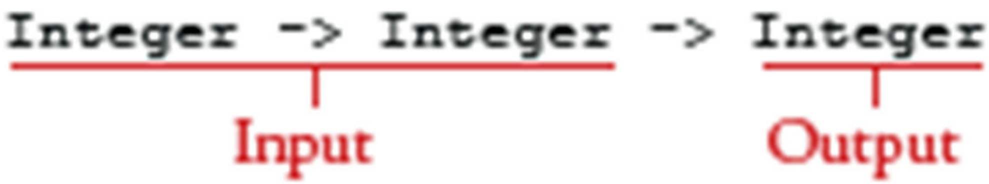
Some “partially applicable” devices

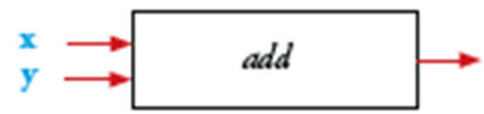


- A partially applicable function is a function that given its first argument **returns** a new, more specialised, function. If you supply this new function with an argument, you get the final result.
- This is what is meant by **partial function application**: you don’t have to pass all the arguments to a function at once

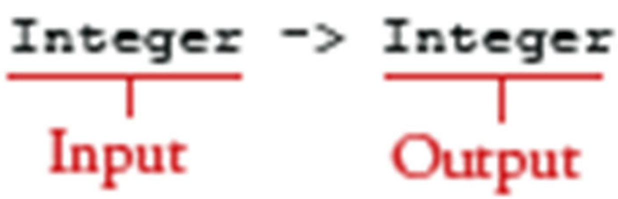
Partial function application returns a function

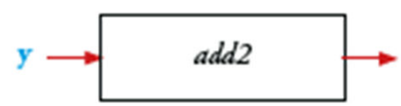
Function type





Integer -> (Integer -> Integer)



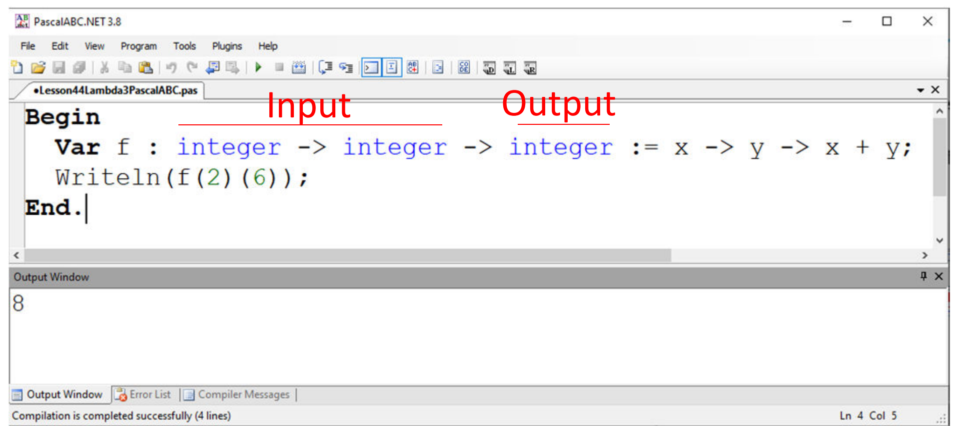


PascalABC.NET example of partial function application

Variable f is assigned a first-class anonymous function that takes two arguments x and y and returns x + y.

Partial function f(2) is a specialised function which takes a single argument and returns a new function which adds 2 to y, i.e. returns 2 + y.

For example, we can apply the partial function f(2) to argument 6, i.e. call f(2)(6) and get 8 returned:



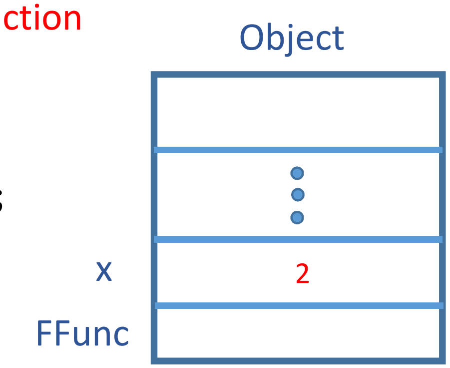
- First class functions can
    - appear in expressions
    - be assigned to a variable
    - Be passed as an argument to another function
    - Be returned as the result of a function call

Delphi example of partial function application

```
Program DelphiAdd2Project;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Type
  TFunctionOfInteger = Reference To Function(Parameter : Integer) : Integer;
Function f(x : Integer) : TFunctionOfInteger;
  Begin
    Result := Function(y : Integer) : Integer
              Begin
                Result := x + y;
              End;
  End;
Begin
  Writeln(f(2)(6));
  Readln;
End.
```

TFunc is a reference to a generic function

Could use TFunc<Integer, Integer>



In computer science, a **closure** is a first-class function with free variables that are bound in the lexical environment, i.e. a closure captures variables from its surrounding scope at define-time and is able to use these variables at execution time even if these variables are no longer in scope, i.e. closures preserve the outer scope inside an inner scope. Closures only make a function impure if you modify the closed-over variable.

Can use functional programming style in OOP to gain benefits of FP

- Can use functional programming style in OOP to gain benefits of FP:
  - ❑ Testability because the result returned by a pure function depends only upon the arguments passed into it, and because the function generates no side-effects, automated tests are easier to write and more effective.
  - ❑ Provability. If functions A and B are pure, side-effect free functions, and both A and B are correct, then any combination of A and B is also correct. This is not true when combining functions and methods that do not adopt this pure approach.
  - ❑ Parallelism. Functionality written using the pure FP approach is much easier to parallelise for performance and scalability.
- If the function is only used once, or a limited number of times, an anonymous function may be syntactically lighter than using a named function.
- Anonymous functions are ubiquitous in functional programming languages and other languages with first-class functions.
- Using pure functions simplifies parallel computing since two purely functional parts of the evaluation never interact, and a function always returns the same output, given the same input.

Higher-order functions

The next example explores anonymous functions/procedures in the context of higher-order functions.

A function that takes a function as an argument or returns a function as a result (or does both) is a higher-order function.

Higher-order functions make it possible to define very general functions that are useful in a variety of applications.

Map

Our first example of a higher-order function is the map function.
This function applies a given function to each element of a list, returning a list of results.
For example, to apply the action of squaring to every element of an array of integer [1,2,3,4,5,6,7,8,9,10] we do the following

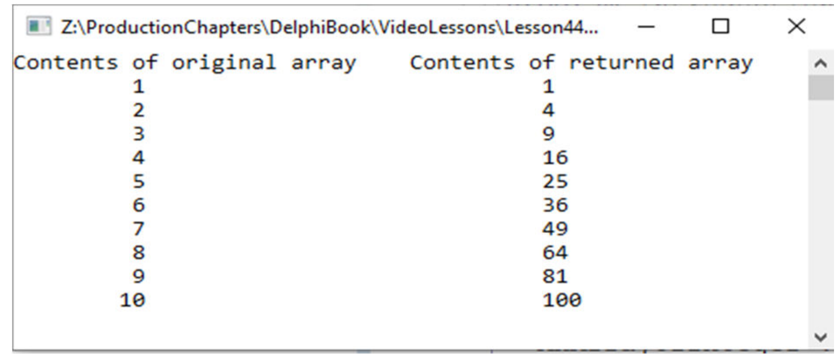
map f [1,2,3,4,5,6,7,8,9,10]

map f [1,2,3,4,5,6,7,8,9,10]
means [f 1, f 2, ..., f 10]

```
Map(Function(x : Integer) : Integer
      Begin
        Result := x * x;
      End, AnArrayOfInteger);
```

Map function that uses an anonymous function

```
Program IntroductionToMapProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Type
  TFunctionOfInteger = Reference to Function(x : Integer) : Integer;
  TArrayOfInteger = Array Of Integer;
Var
  AnArrayOfInteger : TArrayOfInteger = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  AnotherArrayOfInteger : TArrayOfInteger;
Function Map(f : TFunctionOfInteger; ArrayOfInteger : TArrayOfInteger) : TArrayOfInteger;
  Begin
    SetLength(Result, Length(ArrayOfInteger)); {Result variable stores reference to array block of memory}
    For Var i := Low(ArrayOfInteger) To High(ArrayOfInteger)
      Do Result[i] := f(ArrayOfInteger[i]);
  End;
Begin
  AnotherArrayOfInteger := Map(Function(x : Integer) : Integer
                               Begin
                                 Result := x * x;
                               End, AnArrayOfInteger);
  Writeln('Contents of original array    Contents of returned array');
  For Var i :=  Low(AnotherArrayOfInteger) To High(AnotherArrayOfInteger)
    Do Writeln(AnArrayOfInteger[i] : 10, ' ' : 30, AnotherArrayOfInteger[i]);
  Readln;
End.
```



Reduce function

Reduce or fold

Reduce or fold is the name of a higher-order function which reduces a list of values to a single value by repeatedly applying a combining function to the list of values.

In the folding or reduction process, a function, e.g. sum, is applied to the list, element by element, returning something such as the total sum of all elements.

A reduce/fold takes a binary function (function of two variables), a starting value (often called an accumulator), and a list to fold up. The fold reduces the entire list down to a single accumulator value.

reduce f 0 [1,2,3,4,5,6,7,8,9,10]

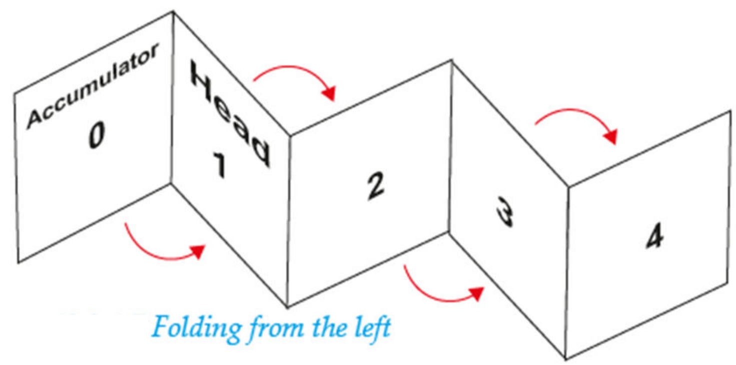
```
Reduce(Function(x, y : Integer) : Integer
         Begin
           Result := x + y;
         End, 0, AnArrayOfInteger);
```

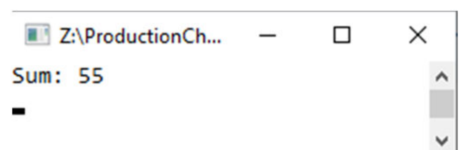
Reduce function using anonymous function

```
Program IntroductionToReduceProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Type
  TFunctionOfInteger = Reference to Function(x, y : Integer) : Integer;
  TArrayOfInteger = Array Of Integer;
Var
  AnArrayOfInteger : TArrayOfInteger = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

Function Reduce(f : TFunctionOfInteger; InitialValue : Integer;
                ArrayOfInteger : TArrayOfInteger) : Integer;
  Begin
    Result := InitialValue; {Result performs role of accumulator}
    For Var i := Low(ArrayOfInteger) To High(ArrayOfInteger)
      Do Result := f(ArrayOfInteger[i], Result);
  End;
Begin
  Var Sum := Reduce(Function(x, y : Integer) : Integer
                    Begin
                      Result := x + y;
                    End, 0, AnArrayOfInteger);
  Writeln('Sum: ', Sum);
  Readln;
End.
```

reduce f 0 [1,2,3,4,5,6,7,8,9,10]





Filter function

The filter function is a higher-order function that processes a data structure, e.g. a list, in some order to produce a new data structure containing exactly those elements of the original data structure that match a given condition.

For example, filter can apply the even function to every element of the integer list [1,2,3,4,5,6,7,8,9,10] and return a list containing integers that possess the property of evenness.
E.g

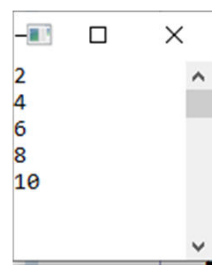
filter even [1,2,3,4,5,6,7,8,9,10]

```
Filter(Function (Item : Integer) : Boolean
         Begin
           Result := Item Mod 2 = 0; {Tests for evenness}
         End, AnArrayOfInteger);
```

Filter function using anonymous function

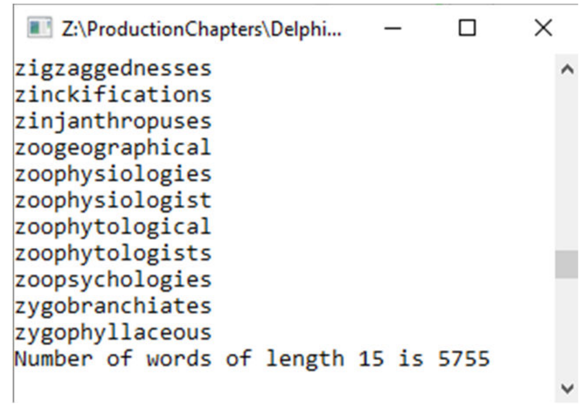
```
Program IntroductionToFilterProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils;
Type
  TFunctionOfInteger = Reference to Function(x : Integer) : Boolean;
  TArrayOfInteger = Array Of Integer;
Var
  AnArrayOfInteger : TArrayOfInteger = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  AnotherArrayOfInteger : TArrayOfInteger;

Function Filter(f : TFunctionOfInteger; ArrayOfInteger : TArrayOfInteger) : TArrayOfInteger;
  Var Count, NewLength : Integer;
  Begin
    Count := -1; NewLength := 0;
    For Var i := Low(ArrayOfInteger) To High(ArrayOfInteger)
      Do
        If f(ArrayOfInteger[i]) {Filter operation}
          Then
            Begin
              Inc(Count); Inc(NewLength);
              SetLength(Result, NewLength);{Extends the array's length}
              Result[Count] := ArrayOfInteger[i]; {Adds the item that passes the test}
            End;
  End;
Begin
  AnotherArrayOfInteger := Filter(Function (Item : Integer) : Boolean
                                    Begin
                                      Result := Item Mod 2 = 0; {Tests for evenness}
                                    End, AnArrayOfInteger);
  For Var i :=  Low(AnotherArrayOfInteger) To High(AnotherArrayOfInteger)
    Do Writeln(AnotherArrayOfInteger[i]);
  Readln;
End.
```



Scrabble example of use of Filter higher-order function

```
Program WordsProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses System.SysUtils, System.Classes;
Type
  TFunctionOfString = Reference to Function(AString : String) : Boolean;
Function Filter(GivenWordList : TStringList ; f : TFunctionOfString) : TStringList;
  Begin
    Var FilteredNewWordList := TStringList.Create;
    For Var Word In GivenWordList
      Do If f(Word) Then FilteredNewWordList.Add(Word);
    Result := FilteredNewWordList;
  End;
Var
  WordLength : Integer;
Begin
  Var WordList := TStringList.Create;
  WordList.LoadFromFile('../../Sowpods.txt');
  Write('Input word length to filter on: ');
  Readln(WordLength);
  Var NewWordList := Filter(WordList, {with} Function (Word : String) : Boolean
                                               Begin
                                                 If Word.Length = WordLength
                                                   Then Result := True
                                                   Else Result := False;
                                               End);
  If NewWordList.Count <> 0
    Then
      Begin
        For Var Word In NewWordList Do Writeln(Word);
        Writeln('Number of words of length ', WordLength, ' is ', NewWordList.Count);
      End
    Else Writeln('No words of length ', WordLength);
  Readln;
End.
```



Using an anonymous function makes it very easy to change the filter criterion

Programs used in Session 34 Delphi Boot Camp 2022
And some that there wasn't time to demonstrate

LearnDelphi.org

- UKDevelopersProgs
  - PolynomialSumProject.exe
  - AnonymousMethodSquareProject.exe
  - DelphiAdd2Project.exe
  - IntroductionToMapProject.exe
  - IntroductionToReduceProject.exe
  - IntroductionToFilterProject.exe
  - WordsProject.exe
  - ClosuresExplainedProject.exe
  - ClosuresExplainedProjectArgumentCounter.exe
  - BirdSongsProject.exe
  - EmulatingAnonymousMethodsProject.exe
  - MemoizeProject.exe
  - ReflectionProject.exe
  - UsingRTTIForAdd6Project.exe
  - vmt_viewer.exe
  - vmt_viewerWithConsole.exe
  - TArrayHelperProject.exe
  - StandardisingMethodProcedureSignatureProject.exe
  - StandardisingMethodProcedureSignatureElaboratedProject.exe
  - AnonAjaxHTTPS.exe
  - UsingClickProcedureProject.exe
  - EventHandlerWithAnonymousMethodProject.exe
  - NonParallelPrimes.exe
  - TwelveTimesTableProject.exe
  - TTaskRun2Project.exe
  - SimplifiedIteratorProject.exe
  - TwelveTimesTableWithGenericArrayProject.exe
  - **ParallelVersusNonParallelPrimes.exe**

Contact drbond@educational-computing.co.uk
To obtain a download link to the source code for these programs

Or
Use download link provided by Embarcadero at LearnDelphi.org.

Exploiting parallelism using an anonymous procedure

```
Const
  UpperInteger = 10000000;

Begin
   Try
     // counts the prime numbers below a given value using a single thread
     Total := 0;
     StopWatch :=TStopWatch.Create;
     StopWatch.Start;
     For Var i : Int64 := 2 To UpperInteger
       Do
         If IsPrime(i)
           Then Total := Total + 1;
     StopWatch.Stop;
     Writeln(Format('Non-parallel For loop. Time (in milliseconds): %d - Primes found: %d', [StopWatch.ElapsedMilliseconds,Total]));
     // counts the prime numbers below a given value using parallelisation of loop
     Total := 0;
     StopWatch :=TStopWatch.Create;
     StopWatch.Start;
     TParallel.For(2, UpperInteger, Procedure(i : Int64)
                                     Begin
                                       If IsPrime(i)
                                         Then TInterlocked.Increment(Total);
                                     End);
     StopWatch.Stop;
     Writeln(Format('Parallel For loop. Time (in milliseconds): %d - Primes found: %d', [StopWatch.ElapsedMilliseconds,Total]));
     Readln;
   Except On E : EAggregateException
     Do Writeln(E.ToString);
   End;
End.
```

```
Function IsPrime(n : Int64) : Boolean;
  Begin
    Var k : Int64 := Trunc(Sqrt(n));
    Var i : Int64 := 2;
    While  (i <= k) And ((n Mod i) <> 0)
      Do Inc(i);
    Result := i > k;
End;
```

Simplifies parallel computing since two purely functional parts of the evaluation never interact, and a function always returns the same output, given the same input.

Multiple threads: Each thread tests a different part of the range 2..10000000. Lost update problem avoided by locking variable Total in order to enable exclusive access when updating.

Exploiting parallelism using an anonymous procedure

AMD Ryzen 5 3600 6-core processor

Console output

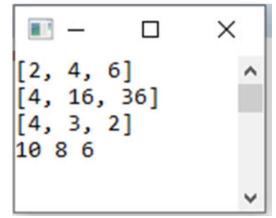
```
Z:\ProductionChapters\DelphiBook\VideoLessons\Lesson44\ParallelVersusNonParallelPrimes\Win32\Debug\Parallel...
Non-parallel For loop. Time (in milliseconds): 8608 - Primes found: 664579
Parallel For loop. Time (in milliseconds): 1449 - Primes found: 664579
```

Console output shows that the non-parallel solution takes roughly six times longer to execute than the parallel solution.

Putting together a fluent interface support unit

```
Unit TArrayHelperUnit;
Interface
Uses
  System.SysUtils,
  Math, System.Generics.Collections;
Type
  TArrayHelper = Record Helper For TArray<Double>
                   Strict Private
                     Type
                       TForEachRef = Reference To Procedure(Var x : Double);
                       TMapRef = TFunc<Double, Double>;
                       TFilterRef = TFunc<Double, Integer, Boolean>;
                       TPredicateRef = TFunc<Double, Boolean>;
                       TReduceRef = TFunc<Double, Double, Boolean>;
                   Public
                     Function ToString : String;
                     Procedure ForEach(Lambda : TForEachRef);
                     Function Map(Lambda : TMapRef) : TArray<Double>;
                     Function Filter(Lambda : TFilterRef) : TArray<Double>;
                     Function Every(Lambda : TPredicateRef) : Boolean;
                     Function Some(Lambda : TPredicateRef) : Boolean;
                     Function Reduce(Lambda : TReduceRef) : Double; Overload;
                     Function Reduce(Init : Double; Lambda : TReduceRef) : Double; Overload;
                 End;
Implementation
 ……………
End.
```

Putting together a fluent interface



Using closures and higher-order functions in Delphi (habr.com) Use To Google Translate in Firefox

https://sudonull.com/post/103897-Using-Anonymous-Methods-in-Delphi English translation

```
Program TArrayHelperProject;
{$APPTYPE CONSOLE}
{$R *.res}
Uses TArrayHelperUnit in 'TArrayHelperUnit.pas';
Var
  DbleArr1 : TArray<Double> = [1, 2, 3];
  DbleArr2 : TArray<Double> = [5, 4, 3, 2, 1];
  MiddleValuessOfADoubleArray : TArray<Double>;
Begin
  DbleArr1.ForEach(Procedure(Var ValueInArray : Double) Begin ValueInArray := ValueInArray * 2 End);
  Writeln(DbleArr1.ToString);
```

DbleArr1.Map(\ValueInArray -> ValueInArray * ValueInArray)

```
  Var NewDbleArray  := DbleArr1.Map(Function(ValueInArray : Double) : Double
                                    Begin Result := ValueInArray * ValueInArray End);
  Writeln(NewDbleArray.ToString);

  MiddleValuesOfADoubleArray := DbleArr2.Filter(Function(ValueInArray : Double; i : Integer) : Boolean
                                                Begin Result := (ValueInArray > 1) And (ValueInArray < 5) End);
  Writeln(MiddleValuesOfADoubleArray.ToString);

  DbleArr2.Map(Function(ValueInArray : Double) : Double
                         Begin
                           Result := ValueInArray + 5;
                         End).Filter(Function(ValueInArray : Double; Index : Integer) : Boolean
                                       Begin
                                         Result := (Index Mod 2) = 0;
                                       End).ForEach(Procedure(Var ValueInArray : Double)
                                                      Begin Write(ValueInArray : 2 : 0) End);
  Readln;
End.
```

```
Type
  TFunctionOfInteger = Reference To Function(Value : Integer): Integer;
  //Compiler generates an interface TFunctionOfInteger with a single method Invoke with same signature as function
  //An interface is a class with no implementation,
  //E.g. TFunctionOfInteger = Interface Function Invoke(Value : Integer) : Integer; End;
Begin
  Var Square : TFunctionOfInteger := Function(x : Integer) : Integer
                                     Begin
                                       Result := x * x;
                                     End;
  Writeln(Square(6)); {Function application}
```

When Square(6) *is encountered, class* **TFunctionOfInteger$ActRec** *is created and its constructor called which creates an object with a single method* **Invoke** *with same signature as* the anonymous function assigned to Square. **Invoke** *is then called with argument* 6. **Invoke** *then calls the anonymous function assigned to* Square, passing it argument 6.

```
Type
  TIntFunc = Function(Value : Integer): Integer; // Function pointer type
  TFunctionOfInteger$ActRec = Class(TInterfacedObject, TFunctionOfInteger)
                                Private
                                  FFunc : TIntFunc; //Function pointer
                                  Function Invoke(Value : Integer) : Integer;
                                Public
                                  Constructor Create(AFunc : TIntFunc);
                              End;
Constructor TFunctionOfInteger$ActRec.Create(AFunc : TIntFunc);
  Begin
    FFunc:= AFunc;
  End;

Function TFunctionOfInteger$ActRec.Invoke(Value : Integer) : Integer;
  Begin
    Result:= FFunc(Value);
  End;
```

```
AnonMethodObjectReference := TFunctionOfInteger$ActRec.Create(Square);
AnonMethodObjectReference.Invoke(6);
```

SOLID Principles of Object-Oriented Design

S: Single responsibility principle.
O: Open–closed principle.
L: Liskov substitution principle.
I: Interface segregation principle.
D: Dependency inversion principle.

Single Responsibility Principle: Every class, module, or function in a program should have one responsibility/purpose in a program. Or "every class should have only one reason to change".

Interface Segregation Principle: The interface of a program should be split in a way that the user/client would only have access to the necessary methods related to their needs.

Combining these two principles in extremis then

Every interface should have only one method.

An interface with only one method is just a function type.
E.g.

```
TFunctionOfInteger = Reference To Function(Value : Integer): Integer;
```

A bit of theory

- What is a function?
- What is meant by function type?
- What is meant by function application?
- What is meant by a first class object?
- What is meant by partial function application?
- What is Lambda calculus?

Function

- Loosely speaking, a function is a rule that,
  - ❑ for each element in some set A of inputs,
  - ❑ assigns an output chosen from set B but without necessarily using every member of B.

- For example, the function f

$$f : \{0,1,2,3\} \to \{0,1,2,3,4,5,6,7,8,9\}$$

maps 0 to 0, 1 to 1, 2 to 4 and 3 to 9 when the rule is:

output the square of the input.

{0,1,2,3} corresponds to set A.

{0,1,2,3,4,5,6,7,8,9} corresponds to set B.

Function type

- Just as data values (e.g. 6, 9.1, True) have types (integer, real, Boolean respectively) so do functions.

- Function types are important because they state what type of argument a function requires and what type of result it will return.

- A function f which takes an argument of type A and returns a result of type B has a function type which is written

$$A \rightarrow B$$

- To state that f has this type, we write

$$f : A \rightarrow B$$

If f : A → B is a function from A to B we call the set A, the domain of f, and the set B the co-domain of f.

A -> B is a function type. The function f has the function type A -> B or type signature A -> B

Function type

- For example,

  1) squareroot : real → real

  2) square : integer → integer

- The function named squareroot applied to an argument of data type real, produces a result of data type real,

  e.g.

  squareroot (4.0) → 2.0

Functional programming paradigm

- An anonymous function that squares its input is written in Haskell as

  (\x -> x*x)

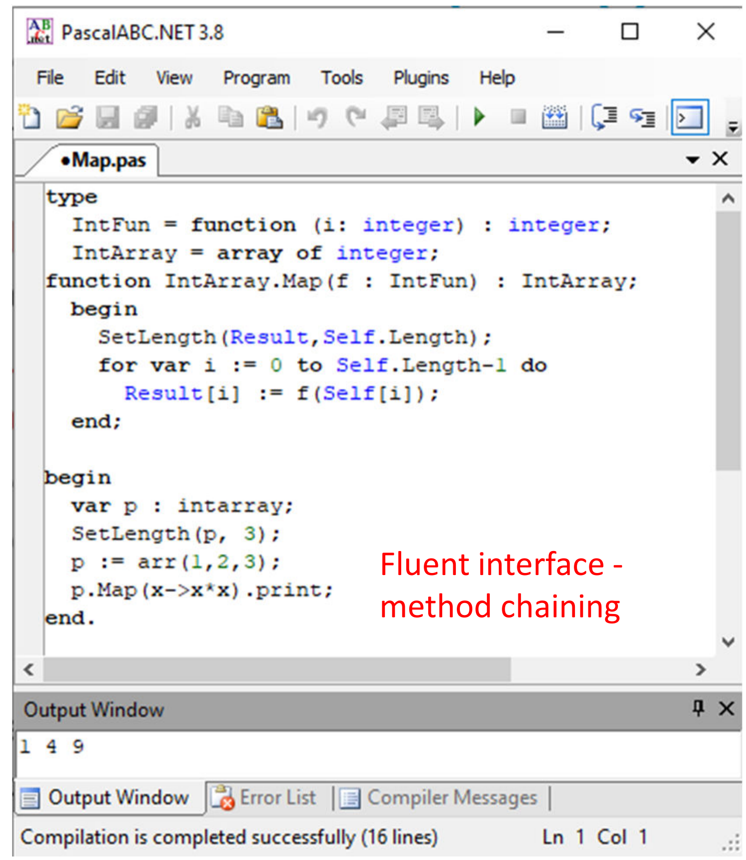
- To apply this function to a list of numbers [1, 2, 3] we can use the built-in map function of Haskell in the following function composition

  map (\x -> x*x) [1,2,3]

- PascalABC.NET is an imperative, procedural and object-oriented language that also supports programming in a functional programming style using LINQ

Under the hood though, all LINQ queries are translated into a set of query expressions. These typically involve a heavy use of lambda expressions and closures.

Garbage collection is considered essential to functional programming – PascalABC.Net as its name suggests relies on .NET. Pure FP uses a lot of recursive data structures which would soon exhaust the stack so under the hood it is necessary to make use of the heap. Using the heap means some form of memory management is required to grab and release the allocated memory. Memory management becomes very complex so the use of a memory management system that guards against cycles etc is very helpful.



Using a functional approach in PascalABC.Net